École Polytechnique

INF549

Initiation à OCaml

Jean-Christophe Filliâtre

11 septembre 2018

présentation du cours

cours	• Jean-Christophe Filliâtre
TD	• Stéphane Lengrand
	• lundi 17 et mardi 18, 9h–12h

toutes les infos sur le site web du cours

`http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Christophe.Filliatre/INF549/`

questions ⇒ `Jean-Christophe.Filliatre@lri.fr`

Présentation

OCaml est un langage fonctionnel, fortement typé, généraliste

Successeur de Caml Light (lui-même successeur de « Caml lourd »)
De la famille ML (SML, F#, etc.)

Conçu et implémenté à l'INRIA Rocquencourt par Xavier Leroy et d'autres

Quelques applications : calcul symbolique et langages (IBM, Intel, Dassault Systèmes), analyse statique (Microsoft, ENS), manipulation de fichiers (Unison, MLDonkey), finance (LexiFi, Jane Street Capital), enseignement

premiers pas en OCaml

le premier programme

hello.ml

```
print_string "hello world!\n"
```

compilation

```
% ocamlopt -o hello hello.ml
```

exécution

```
% ./hello
hello world!
```

le premier programme

hello.ml

```
print_string "hello world!\n"
```

compilation

```
% ocamlopt -o hello hello.ml
```

exécution

```
% ./hello
hello world!
```

le premier programme

`hello.ml`

```
print_string "hello world!\n"
```

compilation

```
% ocamlopt -o hello hello.ml
```

exécution

```
% ./hello
hello world!
```

déclarations

programme = suite de déclarations et d'expressions à évaluer

```
let x = 1 + 2;;
print_int x;;
let y = x * x;;
print_int y;;
```

notion de variable

`let` x = e introduit une variable globale

différences avec la notion usuelle de variable :

1. nécessairement initialisée
2. type pas déclaré mais inféré
3. contenu non modifiable

Java	OCaml
`final int x = 42;`	`let x = 42`

références

une variable modifiable s'appelle une référence

elle est introduite avec `ref`

```
let x = ref 1;;
print_int !x;;
x := !x + 1;;
print_int !x;;
```

références

une variable modifiable s'appelle une référence

elle est introduite avec `ref`

```
let x = ref 1;;
print_int !x;;
x := !x + 1;;
print_int !x;;
```

expressions et instructions

pas de distinction expression/instruction dans la syntaxe : que des expressions

constructions usuelles :

- conditionnelle

```
if i = 1 then 2 else 3
```

- boucle for

```
for i = 1 to 10 do x := !x + i done
```

- séquence

```
x := 1; 2 * !x
```

expressions et instructions

pas de distinction expression/instruction dans la syntaxe : que des expressions

constructions usuelles :

- conditionnelle

```
if i = 1 then 2 else 3
```

- boucle for

```
for i = 1 to 10 do x := !x + i done
```

- séquence

```
x := 1; 2 * !x
```

expressions et instructions

pas de distinction expression/instruction dans la syntaxe : que des expressions

constructions usuelles :

- conditionnelle

```
if i = 1 then 2 else 3
```

- boucle `for`

```
for i = 1 to 10 do x := !x + i done
```

- séquence

```
x := 1; 2 * !x
```

expressions et instructions

pas de distinction expression/instruction dans la syntaxe : que des expressions

constructions usuelles :

- conditionnelle

```
if i = 1 then 2 else 3
```

- boucle `for`

```
for i = 1 to 10 do x := !x + i done
```

- séquence

```
x := 1; 2 * !x
```

type unit

les expressions sans réelle valeur (affectation, boucle, ...) ont pour type unit

ce type a une unique valeur, notée ()

c'est le type donné à la branche else lorsqu'elle est absente

correct :

```
if !x > 0 then x := 0
```

incorrect :

```
2 + (if !x > 0 then 1)
```

type unit

les expressions sans réelle valeur (affectation, boucle, ...) ont pour type `unit`

ce type a une unique valeur, notée `()`

c'est le type donné à la branche `else` lorsqu'elle est absente

correct :

```
if !x > 0 then x := 0
```

incorrect :

```
2 + (if !x > 0 then 1)
```

type unit

les expressions sans réelle valeur (affectation, boucle, ...) ont pour type `unit`

ce type a une unique valeur, notée `()`

c'est le type donné à la branche `else` lorsqu'elle est absente

correct :

```
if !x > 0 then x := 0
```

incorrect :

```
2 + (if !x > 0 then 1)
```

variables locales

en C ou Java, la portée d'une variable locale est définie par le bloc :

```
{
  int x = 1;
  ...
}
```

en OCaml, une variable locale est introduite par `let in` :

```
  let x = 10 in x * x
```

comme une variable globale :

- nécessairement initialisée
- type inféré
- immuable
- mais portée limitée à l'expression qui suit le `in`

variables locales

en C ou Java, la portée d'une variable locale est définie par le bloc :

```
{
  int x = 1;
  ...
}
```

en OCaml, une variable locale est introduite par `let in` :

```
  let x = 10 in x * x
```

comme une variable globale :

- nécessairement initialisée
- type inféré
- immuable
- mais portée limitée à l'expression qui suit le `in`

variables locales

en C ou Java, la portée d'une variable locale est définie par le bloc :

```
{
  int x = 1;
  ...
}
```

en OCaml, une variable locale est introduite par `let in` :

```
let x = 10 in x * x
```

comme une variable globale :

- nécessairement initialisée
- type inféré
- immuable
- mais portée limitée à l'expression qui suit le `in`

let in = expression

let x = e1 in e2 est une expression

son type et sa valeur sont ceux de e2,
dans un environnement où x a le type et la valeur de e1

```
let x = 1 in (let y = 2 in x + y) * (let z = 3 in x * z)
```

parallèle

Java	OCaml
`{ int x = 1;`	`let x = ref 1 in`
`x = x + 1;`	`x := !x + 1;`
`int y = x * x;`	`let y = !x * !x in`
`System.out.print(y); }`	`print_int y`

récapitulation

- programme = suite d'expressions et de déclarations
- variables introduites par le mot clé `let` non modifiables
- pas de distinction expression / instruction

la boucle d'interaction

version interactive du compilateur

```
% ocaml
        OCaml version 4.02.3

# let x = 1 in x + 2;;
```

```
- : int = 3
```

```
# let y = 1 + 2;;
```

```
val y : int = 3
```

```
# y * y;;
```

```
- : int = 9
```

fonctions

syntaxe

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- corps = expression (pas de `return`)
- type inféré (types de l'argument `x` et du résultat)

```
# f 4;;
```

```
- : int = 16
```

syntaxe

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- corps = expression (pas de `return`)
- type inféré (types de l'argument `x` et du résultat)

```
# f 4;;
```

```
- : int = 16
```

syntaxe

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- corps = expression (pas de `return`)
- type inféré (types de l'argument `x` et du résultat)

```
# f 4;;
```

```
- : int = 16
```

syntaxe

```
# let f x = x * x;;
```

```
val f : int -> int = <fun>
```

- corps = expression (pas de `return`)
- type inféré (types de l'argument `x` et du résultat)

```
# f 4;;
```

```
- : int = 16
```

parallèle

Java	OCaml

`static int f(int x) {`
`   return x * x;`
`}`
	`let f x =`
	`   x * x`

procédure

une procédure = une fonction dont le résultat est de type `unit`

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

procédure

une procédure = une fonction dont le résultat est de type `unit`

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

procédure

une procédure = une fonction dont le résultat est de type unit

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

procédure

une procédure = une fonction dont le résultat est de type `unit`

```
# let x = ref 1;;
# let set v = x := v;;
```

```
val set : int -> unit = <fun>
```

```
# set 3;;
```

```
- : unit = ()
```

```
# !x;;
```

```
- : int = 3
```

fonction “sans argument”

prend un argument de type unit

```
# let reset () = x := 0;;
```

```
val reset : unit -> unit = <fun>
```

```
# reset ();;
```

fonction "sans argument"

prend un argument de type unit

```
# let reset () = x := 0;;
```

```
val reset : unit -> unit = <fun>
```

```
# reset ();;
```

fonction à plusieurs arguments

```
# let f x y z = if x > 0 then y + x else z - x;;
```

```
val f : int -> int -> int -> int = <fun>
```

```
# f 1 2 3;;
```

```
- : int = 3
```

fonction à plusieurs arguments

```
# let f x y z = if x > 0 then y + x else z - x;;
```

```
val f : int -> int -> int -> int = <fun>
```

```
# f 1 2 3;;
```

```
- : int = 3
```

fonction locale

fonction locale à une expression

```
# let carre x = x * x in carre 3 + carre 4 = carre 5;;
```

```
- : bool = true
```

fonction locale à une autre fonction

```
# let pythagore x y z =
    let carre n = n * n in
    carre x + carre y = carre z;;
```

```
val pythagore : int -> int -> int -> bool = <fun>
```

fonction locale

fonction locale à une expression

```
# let carre x = x * x in carre 3 + carre 4 = carre 5;;
```

```
- : bool = true
```

fonction locale à une autre fonction

```
# let pythagore x y z =
    let carre n = n * n in
    carre x + carre y = carre z;;
```

```
val pythagore : int -> int -> int -> bool = <fun>
```

fonction comme valeur de première classe

fonction = expression comme une autre, introduite par `fun`

```
# fun x -> x+1
```

```
- : int -> int = <fun>
```

```
# (fun x -> x+1) 3;;
```

```
- : int = 4
```

en réalité

```
let f x = x+1;;
```

est la même chose que

```
let f = fun x -> x+1;;
```

fonction comme valeur de première classe

fonction = expression comme une autre, introduite par fun

```
# fun x -> x+1
```

```
- : int -> int = <fun>
```

```
# (fun x -> x+1) 3;;
```

```
- : int = 4
```

en réalité

```
let f x = x+1;;
```

est la même chose que

```
let f = fun x -> x+1;;
```

application partielle

```
fun x y -> x*x + y*y
```

est la même chose que

```
fun x -> fun y -> x*x + y*y
```

on peut appliquer une fonction partiellement

```
# let f x y = x*x + y*y;;
```

```
val f : int -> int -> int = <fun>
```

```
# let g = f 3;;
```

```
val g : int -> int = <fun>
```

```
# g 4;;
```

```
- : int = 25
```

application partielle (suite)

l'application partielle est une manière de renvoyer une fonction

mais on peut aussi renvoyer une fonction à l'issue d'un calcul

```
# let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

```
val f : int -> int -> int = <fun>
```

une application partielle de `f` ne calcule `x*x` qu'une seule fois

application partielle : exemple

```
# let count_from n =
    let r = ref (n-1) in fun () -> incr r; !r;;
```

```
val count_from : int -> unit -> int = <fun>
```

```
# let c = count_from 0;;
```

```
val c : unit -> int = <fun>
```

```
# c ();;
```

```
- : int = 0
```

```
# c ();;
```

```
- : int = 1
```

ordre supérieur

une fonction peut prendre des fonctions en arguments

```
# let integral f =
    let n = 100 in
    let s = ref 0.0 in
    for i = 0 to n-1 do
      let x = float i /. float n in s := !s +. f x
    done;
    !s /. float n
```

```
# integral sin;;
```

```
- : float = 0.455486508387318301
```

```
# integral (fun x -> x*.x);;
```

```
- : float = 0.32835
```

itérateurs

en Java on itère ainsi sur les éléments d'une structure de données

```
for (Elt x: s) {
    ... quelque chose avec x ...
}
```

en OCaml, on écrit typiquement

```
iter (fun x -> ... quelque chose avec x ...) s
```

où `iter` est une fonction fournie par la structure de données, de type

```
val iter: (elt -> unit) -> set -> unit
```

exemple d'utilisation :

```
iter (fun x -> Printf.printf "%s\n" x) s
```

itérateurs

en Java on itère ainsi sur les éléments d'une structure de données

```
for (Elt x: s) {
    ... quelque chose avec x ...
}
```

en OCaml, on écrit typiquement

```
iter (fun x -> ... quelque chose avec x ...) s
```

où `iter` est une fonction fournie par la structure de données, de type

```
val iter: (elt -> unit) -> set -> unit
```

exemple d'utilisation :

```
iter (fun x -> Printf.printf "%s\n" x) s
```

différence avec les pointeurs de fonctions

"en C aussi on peut passer et renvoyer des fonctions par l'intermédiaire de pointeurs de fonctions"

mais les fonctions d'OCaml sont plus que des pointeurs de fonctions

```
let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

la valeur de x2 est retenue dans une **clôture**

note : il y a des clôtures aussi en Java (≥ 8)

```
s.forEach(x -> { System.out.println(x); });
```

différence avec les pointeurs de fonctions

"en C aussi on peut passer et renvoyer des fonctions par l'intermédiaire de pointeurs de fonctions"

mais les fonctions d'OCaml sont plus que des pointeurs de fonctions

```
let f x = let x2 = x * x in fun y -> x2 + y * y;;
```

la valeur de `x2` est retenue dans une **clôture**

note : il y a des clôtures aussi en Java (≥ 8)

```
s.forEach(x -> { System.out.println(x); });
```

fonctions récursives

en OCaml, le recours aux fonctions récursives est naturel, car

- un appel de fonction ne coûte pas cher
- la récursivité terminale est correctement compilée

exemple :

```
let zero f =
  let rec find i = if f i = 0 then i else find (i+1) in
  find 0
```

code récursif $\Rightarrow$ plus lisible, plus simple à justifier

fonctions récursives

en OCaml, le recours aux fonctions récursives est naturel, car

- un appel de fonction ne coûte pas cher
- la récursivité terminale est correctement compilée

exemple :

```
let zero f =
  let rec find i = if f i = 0 then i else find (i+1) in
  find 0
```

code récursif ⇒ plus lisible, plus simple à justifier

polymorphisme

```
# let f x = x;;
val f : 'a -> 'a = <fun>
# f 3;;
- : int = 3
# f true;;
- : bool = true
# f print_int;;
- : int -> unit = <fun>
# f print_int 1;;
1- : unit = ()
```

polymorphisme

```
# let f x = x;;
```

```
val f : 'a -> 'a = <fun>
```

```
# f 3;;
```

```
- : int = 3
```

```
# f true;;
```

```
- : bool = true
```

```
# f print_int;;
```

```
- : int -> unit = <fun>
```

```
# f print_int 1;;
```

```
1- : unit = ()
```

polymorphisme (suite)

OCaml infère toujours le type le plus général possible

exemple :

```
# let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

polymorphisme (suite)

OCaml infère toujours le type le plus général possible

exemple :

```
# let compose f g = fun x -> f (g x);;
```

```
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

récapitulation

- fonctions = valeurs comme les autres : locales, anonymes, arguments d'autres fonctions, etc.
- partiellement appliquées
- polymorphes
- l'appel de fonction ne coûte pas cher

allocation mémoire

GC

allocation mémoire réalisée par un garbage collector (GC)

intérêts :

- récupération automatique
- allocation efficace

⇒ perdre le réflexe « allouer dynamiquement coûte cher »
... mais continuer à se soucier de complexité !

GC

allocation mémoire réalisée par un garbage collector (GC)

intérêts :

- récupération automatique
- allocation efficace

$\Rightarrow$ perdre le réflexe « allouer dynamiquement coûte cher »
... mais continuer à se soucier de complexité !

tableaux

```
# let a = Array.create 10 0;;
```

```
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

nécessairement initialisé

```
# let a = [| 1; 2; 3; 4 |];;
```

```
# a.(1);;
```

```
- : int = 2
```

```
# a.(1) <- 5;;
```

```
- : unit = ()
```

```
# a;;
```

```
- : int array = [|1; 5; 3; 4|]
```

tableaux

```
# let a = Array.create 10 0;;
```

```
val a : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

nécessairement initialisé

```
# let a = [| 1; 2; 3; 4 |];;
```

```
# a.(1);;
```

```
- : int = 2
```

```
# a.(1) <- 5;;
```

```
- : unit = ()
```

```
# a;;
```

```
- : int array = [|1; 5; 3; 4|]
```

parallèle

Java	OCaml
`int[] a = new int[42];`	`let a = Array.make 42 0`
`a[17]`	`a.(17)`
`a[7] = 3;`	`a.(7) <- 3`
`a.length`	`Array.length a`

tri par insertion

```
let insertion_sort a =
  let swap i j =
    let t = a.(i) in a.(i) <- a.(j); a.(j) <- t
  in
  for i = 1 to Array.length a - 1 do
    (* insérer l'élément a[i] dans a[0..i-1] *)
    let j = ref (i - 1) in
    while !j >= 0 && a.(!j) > a.(!j + 1) do
      swap !j (!j + 1); decr j
    done
  done
```

tri par insertion

```
let insertion_sort a =
  let swap i j =
    let t = a.(i) in a.(i) <- a.(j); a.(j) <- t
  in
  for i = 1 to Array.length a - 1 do
    (* insérer l'élément a[i] dans a[0..i-1] *)
    let rec insert j =
      if j >= 0 && a.(j) > a.(j+1) then
      begin swap j (j+1); insert (j-1) end in
    insert (i-1)
  done
```

on déclare le type enregistrement

```
type complexe = { re : float; im : float }
```

allocation et initialisation simultanées :

```
# let x = { re = 1.0; im = -1.0 };;
```

```
val x : complexe = {re = 1.; im = -1.}
```

```
# x.im;;
```

```
- : float = -1.
```

enregistrements

on déclare le type enregistrement

```
type complexe = { re : float; im : float }
```

allocation et initialisation simultanées :

```
# let x = { re = 1.0; im = -1.0 };;
```

```
val x : complexe = {re = 1.; im = -1.}
```

```
# x.im;;
```

```
- : float = -1.
```

champs modifiables en place

```
type person = { name: string; mutable age: int }
```

```
# let p =  { name = "Martin"; age = 23 };;
```

```
val p : person = {name = "Martin"; age = 23}
```

```
# p.age <- p.age + 1;;
```

```
- : unit = ()
```

```
# p.age;;
```

```
- : int = 24
```

champs modifiables en place

```
type person = { name: string; mutable age: int }
```

```
# let p =  { name = "Martin"; age = 23 };;
```

```
val p : person = {name = "Martin"; age = 23}
```

```
# p.age <- p.age + 1;;
```

```
- : unit = ()
```

```
# p.age;;
```

```
- : int = 24
```

parallèle

Java	OCaml

`class T {`
`   final int v; boolean b;`
`   T(int v, boolean b) {`
`      this.v = v; this.b = b;`
`   }`
`}`
	`type t = {`
	`   v: int;`
	`   mutable b: bool;`
	`}`

`T r = new T(42, true);`	`let r = { v = 42; b = true }`
`r.b = false;`	`r.b <- false`
`r.v`	`r.v`

retour sur les références

référence = enregistrement du type suivant

```
type 'a ref = { mutable contents : 'a }
```

ref, ! et := ne sont que du sucre syntaxique

seuls les tableaux et les champs déclarés mutable sont modifiables en place

retour sur les références

référence = enregistrement du type suivant

```
type 'a ref = { mutable contents : 'a }
```

`ref`, `!` et `:=` ne sont que du sucre syntaxique

seuls les tableaux et les champs déclarés `mutable` sont modifiables en place

retour sur les références

référence = enregistrement du type suivant

```
type 'a ref = { mutable contents : 'a }
```

`ref`, `!` et `:=` ne sont que du sucre syntaxique

seuls les tableaux et les champs déclarés mutable sont modifiables en place

n-uplets

notation usuelle

```
# (1,2,3);;
```

```
- : int * int * int = (1, 2, 3)
```

```
# let v = (1, true, "bonjour", 'a');;
```

```
val v : int * bool * string * char =
  (1, true, "bonjour", 'a')
```

accès aux éléments

```
# let (a,b,c,d) = v;;
```

```
val a : int = 1
val b : bool = true
val c : string = "bonjour"
val d : char = 'a'
```

n-uplets

notation usuelle

```
# (1,2,3);;
```

```
- : int * int * int = (1, 2, 3)
```

```
# let v = (1, true, "bonjour", 'a');;
```

```
val v : int * bool * string * char =
  (1, true, "bonjour", 'a')
```

accès aux éléments

```
# let (a,b,c,d) = v;;
```

```
val a : int = 1
val b : bool = true
val c : string = "bonjour"
val d : char = 'a'
```

n-uplets (suite)

exemple d'utilisation

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;
```

```
val division : int -> int -> int * int = <fun>
```

fonction prenant un n-uplet en argument

```
# let f (x,y) = x + y;;
```

```
val f : int * int -> int = <fun>
```

```
# f (1,2);;
```

```
- : int = 3
```

n-uplets (suite)

exemple d'utilisation

```
# let rec division n m =
    if n < m then (0, n)
    else let (q,r) = division (n - m) m in (q + 1, r);;
```

```
val division : int -> int -> int * int = <fun>
```

fonction prenant un n-uplet en argument

```
# let f (x,y) = x + y;;
```

```
val f : int * int -> int = <fun>
```

```
# f (1,2);;
```

```
- : int = 3
```

listes

type prédéfini de listes, α `list`, immuables et homogènes
construites à partir de la liste vide `[]` et de l'ajout en tête `::`

```
# let l = 1 :: 2 :: 3 :: [];;
```

```
val l : int list = [1; 2; 3]
```

ou encore

```
# let l = [1; 2; 3];;
```

filtrage

filtrage = construction par cas sur la forme d'une liste

```
# let rec sum l =
    match l with
    | [] -> 0
    | x :: r -> x + sum r;;
```

```
val sum: int list -> int = <fun>
```

```
# sum [1;2;3];;
```

```
- : int = 6
```

notation plus compacte pour une fonction filtrant son argument

```
let rec sum = function
  | [] -> 0
  | x :: r -> x + sum r;;
```

filtrage

filtrage = construction par cas sur la forme d'une liste

```
# let rec sum l =
    match l with
    | [] -> 0
    | x :: r -> x + sum r;;
```

```
val sum: int list -> int = <fun>
```

```
# sum [1;2;3];;
```

```
- : int = 6
```

notation plus compacte pour une fonction filtrant son argument

```
let rec sum = function
  | [] -> 0
  | x :: r -> x + sum r;;
```

filtrage

filtrage = construction par cas sur la forme d'une liste

```
# let rec sum l =
    match l with
    | [] -> 0
    | x :: r -> x + sum r;;
```

```
val sum: int list -> int = <fun>
```

```
# sum [1;2;3];;
```

```
- : int = 6
```

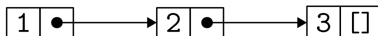
notation plus compacte pour une fonction filtrant son argument

```
let rec sum = function
  | [] -> 0
  | x :: r -> x + sum r;;
```

représentation mémoire

listes OCaml = mêmes listes chaînées qu'en C ou Java

la liste `[1; 2; 3]` correspond à



types construits

listes = cas particulier de types construits

type construit = réunion de plusieurs constructeurs

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

listes définies par

```
type 'a list = [] | :: of 'a * 'a list
```

types construits

listes = cas particulier de types construits

type construit = réunion de plusieurs constructeurs

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

listes définies par

```
type 'a list = [] | :: of 'a * 'a list
```

types construits

listes = cas particulier de types construits

type construit = réunion de plusieurs constructeurs

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

listes définies par

```
type 'a list = [] | :: of 'a * 'a list
```

types construits

listes = cas particulier de types construits

type construit = réunion de plusieurs constructeurs

```
type fmla = True | False | And of fmla * fmla
```

```
# True;;
```

```
- : fmla = True
```

```
# And (True, False);;
```

```
- : fmla = And (True, False)
```

listes définies par

```
type 'a list = [] | :: of 'a * 'a list
```

filtrage

le filtrage se généralise

```
# let rec eval = function
    | True -> true
    | False -> false
    | And (f1, f2) -> eval f1 && eval f2;;
```

```
val eval: fmla -> bool = <fun>
```

filtrage

les motifs peuvent être imbriqués :

```
let rec eval = function
  | True -> true
  | False -> false
  | And (False, f2) -> false
  | And (f1, False) -> false
  | And (f1, f2) -> eval f1 && eval f2;;
```

filtrage

les motifs peuvent être omis ou regroupés

```
let rec eval = function
  | True -> true
  | False -> false
  | And (False, _) | And (_, False) -> false
  | And (f1, f2) -> eval f1 && eval f2;;
```

parallèle

Java

```
abstract class Fmla { }
class True extends Fmla { }
class False extends Fmla { }
class And extends Fmla {
  Fmla f1, f2; }

abstract class Fmla {
  abstract boolean eval(); }
class True { boolean eval() {
  return true; } }
class False { boolean eval() {
  return false; } }
class And { boolean eval() {
  return f1.eval()&&f2.eval();
} }
```

OCaml

```
type fmla =
| True
| False
| And of fmla * fmla


let rec eval = function
| True -> true

| False -> false

| And (f1, f2) ->
    eval f1 && eval f2
```

filtrage

le filtrage n'est pas limité aux types construits

```
let rec mult = function
  | [] -> 1
  | 0 :: _ -> 0
  | x :: l -> x * mult l
```

on peut écrire `let motif = expression` lorsqu'il y a un seul motif (comme dans `let (a,b,c,d) = v` par exemple)

filtrage

le filtrage n'est pas limité aux types construits

```
let rec mult = function
  | [] -> 1
  | 0 :: _ -> 0
  | x :: l -> x * mult l
```

on peut écrire `let motif = expression` lorsqu'il y a un seul motif (comme dans `let (a,b,c,d) = v` par exemple)

récapitulation

- allouer ne coûte pas cher
- libération automatique
- valeurs allouées nécessairement initialisées
- majorité des valeurs non modifiables en place (seuls tableaux et champs d'enregistrements `mutable`)
- représentation mémoire des valeurs construites efficace
- filtrage = examen par cas sur les valeurs construites

modèle d'exécution

valeurs

une valeur est

- soit une valeur primitive (entier, flottant, booléen, `[]`, etc.)
- soit un pointeur (vers un tableau, un constructeur comme `And`, etc.)

elle tient sur 64 bits

le mode de passage est **par valeur**

en particulier, aucune valeur n'est jamais copiée en profondeur

c'est **exactement comme en Java**

pas de `null`

en OCaml, il n'y a **pas d'équivalent de `null`**

en particulier, toute valeur est nécessairement initialisée

parfois un peu contraignant, mais le jeu en vaut la chandelle :

> *une expression de type τ dont l'évaluation termine*
> *donne nécessairement une valeur légale du type τ*

on parle de typage fort

pas d'équivalent de `NullPointerException`
(ni de `segmentation fault` comme en C/C++)

deux égalités

l'égalité notée == est l'**égalité physique**,
c'est-à-dire l'égalité des pointeurs ou des valeurs primitives

```
# (1, 2) == (1, 2);;
```

```
- : bool = false
```

c'est la même qu'en Java

l'égalité =, en revanche, est **l'égalité structurelle**,
c'est-à-dire l'égalité en profondeur

```
# (1, 2) = (1, 2);;
```

```
- : bool = true
```

c'est comparable à la méthode `equals` en Java (lorsqu'elle est définie)

exceptions

exceptions

c'est la notion usuelle

une exception peut être levée avec raise

```
let division n m =
  if m = 0 then raise Division_by_zero else ...
```

et rattrapée avec try with

```
try division x y with Division_by_zero -> (0,0)
```

on peut déclarer de nouvelles exceptions

```
exception Error
exception Unix_error of string
```

exceptions

c'est la notion usuelle

une exception peut être levée avec raise

```
let division n m =
  if m = 0 then raise Division_by_zero else ...
```

et rattrapée avec try with

```
try division x y with Division_by_zero -> (0,0)
```

on peut déclarer de nouvelles exceptions

```
exception Error
exception Unix_error of string
```

exceptions

c'est la notion usuelle

une exception peut être levée avec raise

```
  let division n m =
    if m = 0 then raise Division_by_zero else ...
```

et rattrapée avec try with

```
  try division x y with Division_by_zero -> (0,0)
```

on peut déclarer de nouvelles exceptions

```
exception Error
exception Unix_error of string
```

idiome

en OCaml, les exceptions sont utilisées dans la bibliothèque pour signifier un résultat exceptionnel

exemple : `Not_found` pour signaler une valeur absente

```
try
    let v = Hashtbl.find table key in
    ...
with Not_found ->
    ...
```

(là où Java renvoie `null` typiquement)

modules et foncteurs

génie logiciel

lorsque les programmes deviennent gros il faut

- découper en unités (modularité)
- occulter la représentation de certaines données (encapsulation)
- éviter au mieux la duplication de code

en OCaml, ces fonctionnalités sont apportées par les modules

fichiers et modules

chaque fichier est un module

si arith.ml contient

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors on le compile avec

```
% ocamlopt -c arith.ml
```

utilisation dans un autre module main.ml :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

fichiers et modules

chaque fichier est un module

si `arith.ml` contient

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors on le compile avec

```
% ocamlopt -c arith.ml
```

utilisation dans un autre module `main.ml` :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

fichiers et modules

chaque fichier est un module

si `arith.ml` contient

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors on le compile avec

```
% ocamlopt -c arith.ml
```

utilisation dans un autre module `main.ml` :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

fichiers et modules

chaque fichier est un module

si `arith.ml` contient

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors on le compile avec

```
% ocamlopt -c arith.ml
```

utilisation dans un autre module `main.ml` :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

fichiers et modules

chaque fichier est un module

si `arith.ml` contient

```
let pi = 3.141592
let round x = floor (x +. 0.5)
```

alors on le compile avec

```
% ocamlopt -c arith.ml
```

utilisation dans un autre module `main.ml` :

```
let x = float_of_string (read_line ());;
print_float (Arith.round (x /. Arith.pi));;
print_newline ();;
```

```
% ocamlopt -c main.ml
```

```
% ocamlopt arith.cmx main.cmx
```

encapsulation

on peut restreindre les valeurs exportées avec une interface

dans un fichier arith.mli

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation

on peut restreindre les valeurs exportées avec une interface
dans un fichier arith.mli

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation

on peut restreindre les valeurs exportées avec une interface dans un fichier `arith.mli`

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation

on peut restreindre les valeurs exportées avec une interface
dans un fichier `arith.mli`

```
val round : float -> float
```

```
% ocamlopt -c arith.mli
% ocamlopt -c arith.ml
```

```
% ocamlopt -c main.ml
File "main.ml", line 2, characters 33-41:
Unbound value Arith.pi
```

encapsulation (suite)

une interface peut restreindre la visibilité de la définition d'un type

dans set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

mais dans set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

le type t est un type abstrait

encapsulation (suite)

une interface peut restreindre la visibilité de la définition d'un type

dans set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

mais dans set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

le type t est un type abstrait

encapsulation (suite)

une interface peut restreindre la visibilité de la définition d'un type
dans set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

mais dans set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

le type t est un type abstrait

encapsulation (suite)

une interface peut restreindre la visibilité de la définition d'un type

dans set.ml

```
type t = int list
let empty = []
let add x l = x :: l
let mem = List.mem
```

mais dans set.mli

```
type t
val empty : t
val add : int -> t -> t
val mem : int -> t -> bool
```

le type t est un type abstrait

compilation séparée

la compilation d'un fichier ne dépend que des interfaces des fichiers utilisés
$\Rightarrow$ moins de recompilation quand un code change mais pas son interface

compilation séparée

la compilation d'un fichier ne dépend que des interfaces des fichiers utilisés
$\Rightarrow$ moins de recompilation quand un code change mais pas son interface

langage de modules

modules non restreints aux fichiers

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

langage de modules

modules non restreints aux fichiers

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

langage de modules

modules non restreints aux fichiers

```
module M = struct
  let c = 100
  let f x = c * x
end
```

```
module A = struct
  let a = 2
  module B = struct
    let b = 3
    let f x = a * b * x
  end
  let f x = B.f (x + 1)
end
```

langage de modules

de même pour les signatures

```
module type S = sig
  val f : int -> int
end
```

contrainte

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

langage de modules

de même pour les signatures

```
module type S = sig
  val f : int -> int
end
```

contrainte

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

langage de modules

de même pour les signatures

```
module type S = sig
  val f : int -> int
end
```

contrainte

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

langage de modules

de même pour les signatures

```
module type S = sig
  val f : int -> int
end
```

contrainte

```
module M : S = struct
  let a = 2
  let f x = a * x
end
```

```
# M.a;;
```

```
Unbound value M.a
```

récapitulation

- modularité par découpage du code en unités appelées modules
- encapsulation de types et de valeurs, types abstraits
- vraie compilation séparée
- organisation de l'espace de nommage

foncteurs

foncteur = module paramétré par un ou plusieurs autres modules

exemple : table de hachage générique

il faut paramétrer par rapport à la fonction de hachage et la fonction d'égalité

la solution : un foncteur

```
module type HashedType = sig
  type elt
  val hash: elt -> int
  val eq  : elt -> elt -> bool
end
```

```
module HashTable(X: HashedType) = struct ... end
```

foncteur : le code

```
module HashTable(X: HashedType) = struct
  type t = X.elt list array
  let create n = Array.create n []
  let add t x =
    let i = (X.hash x) mod (Array.length t) in
    t.(i) <- x :: t.(i)
  let mem t x =
    let i = (X.hash x) mod (Array.length t) in
    List.exists (X.eq x) t.(i)
end
```

foncteur : l'interface

```
module HashTable(X: HashedType) : sig
  type t
  val create: int -> t
  val add: t -> X.elt -> unit
  val mem: t -> X.elt -> bool
end
```

foncteur : utilisation

```
module Entiers = struct
  type elt = int
  let hash x = x land max_int
  let eq x y = x=y
end
```

```
module Hentiers = HashTable(Entiers)
```

```
# let t = Hentiers.create 17;;
```

```
val t : Hentiers.t = <abstr>
```

```
# Hentiers.add t 13;;
```

```
- : unit = ()
```

```
# Hentiers.add t 173;;
```

```
- : unit = ()
```

foncteur : utilisation

```
module Entiers = struct
  type elt = int
  let hash x = x land max_int
  let eq x y = x=y
end
```

```
module Hentiers = HashTable(Entiers)
```

```
# let t = Hentiers.create 17;;
```

```
val t : Hentiers.t = <abstr>
```

```
# Hentiers.add t 13;;
```

```
- : unit = ()
```

```
# Hentiers.add t 173;;
```

```
- : unit = ()
```

foncteur : utilisation

```
module Entiers = struct
  type elt = int
  let hash x = x land max_int
  let eq x y = x=y
end
```

```
module Hentiers = HashTable(Entiers)
```

```
# let t = Hentiers.create 17;;
```

```
val t : Hentiers.t = <abstr>
```

```
# Hentiers.add t 13;;
```

```
- : unit = ()
```

```
# Hentiers.add t 173;;
```

```
- : unit = ()
```

parallèle

Java	OCaml
`interface HashedType<T> {`	`module type HashedType = sig`
	`   type elt`
`   int hash();`	`   val hash: elt -> int`
`   boolean eq(T x);`	`   val eq: elt -> elt -> bool`
`}`	`end`
`class HashTable`	`module HashTable(E: HashedType) =`
`   <E extends HashedType<E>> {`	`struct`
`      ...`	`   ...`

foncteurs : applications

1. structures de données paramétrées par d'autres structures de données
   - `Hashtbl.Make` : tables de hachage
   - `Set.Make` : ensembles finis codés par des arbres équilibrés
   - `Map.Make` : tables d'association codées par des arbres équilibrés

2. algorithmes paramétrés par des structures de données

exemple : algorithme de Dijkstra « générique »

```
module Dijkstra
  (G: sig
        type graph
        type node
        val  succ: graph -> node -> (node * float) list
      end) :
  sig
    val shortest_path:
      G.graph -> G.node -> G.node -> G.node list * float
  end
```

persistance

structures de données immuables

en OCaml, la majorité des structures de données sont **immuables**
(seules exceptions : tableaux et enregistrements à champ `mutable`)

dit autrement :

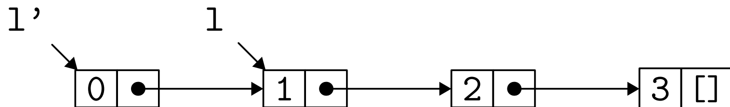
- une valeur n'est pas affectée par une opération,
- mais une **nouvelle** valeur est renvoyée

vocabulaire : on parle de code **purement applicatif** ou encore simplement de **code pur** (parfois aussi de code **purement fonctionnel**)

exemple de structure immuable : les listes

```
let l = [1; 2; 3]
```

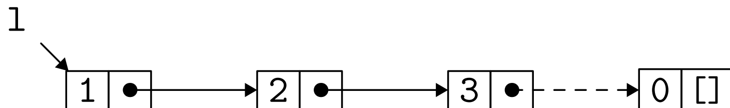
```
let l' = 0 :: l
```



pas de copie, mais partage

contrepartie

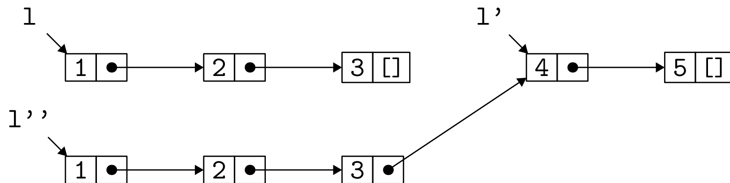
un ajout en queue de liste n'est pas aussi simple :



concaténation de deux listes

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l'
```

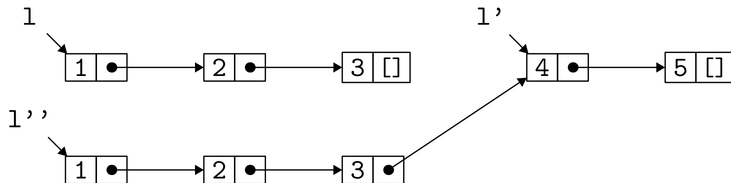


blocs de l copiés, blocs de l' partagés

concaténation de deux listes

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```

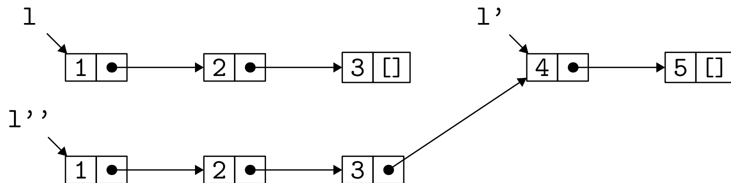


blocs de l copiés, blocs de l' partagés

concaténation de deux listes

```
let rec append l1 l2 = match l1 with
  | [] -> l2
  | x :: l -> x :: append l l2
```

```
let l = [1; 2; 3]
let l' = [4; 5]
let l'' = append l l '
```



blocs de `l` copiés, blocs de `l'` partagés

listes chaînées modifiables en place

note : on peut définir des listes chaînées ≪ traditionnelles ≫,
par exemple ainsi

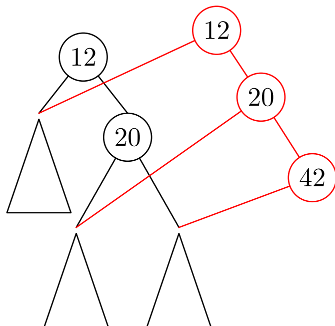
```
type 'a mlist = Empty | Element of 'a element
and 'a element = { value: 'a; mutable next: 'a mlist }
```

mais alors il faut faire attention au **partage** (*aliasing*)

autre exemple : les arbres

```
type tree = Empty | Node of tree * int * tree

val add : int -> tree -> tree
```



là encore, peu de copie et beaucoup de partage

intérêts pratiques de la persistance

1. correction des programmes
    - code plus simple
    - raisonnement mathématique possible

2. outil puissant pour le backtracking
    - algorithmes de recherche
    - manipulations symboliques et portées
    - rétablissement suite à une erreur

intérêts pratiques de la persistance

1. correction des programmes
   - code plus simple
   - raisonnement mathématique possible

2. outil puissant pour le backtracking
   - algorithmes de recherche
   - manipulations symboliques et portées
   - rétablissement suite à une erreur

persistance et backtracking (1)

recherche de la sortie dans un labyrinthe

```
type state
val success: state -> bool
type move
val moves: state -> move list
val move: state -> move -> state
```

```
let rec find s =
  success s || trymove s (moves s)
and trymove s = function
  | []      -> false
  | m :: r -> find (move m s) || trymove s r
```

persistance et backtracking (1)

recherche de la sortie dans un labyrinthe

```
type state
val success: state -> bool
type move
val moves: state -> move list
val move: state -> move -> state
```

```
let rec find s =
  success s || trymove s (moves s)
and trymove s = function
  | []     -> false
  | m :: r -> find (move m s) || trymove s r
```

persistance et backtracking (1)

recherche de la sortie dans un labyrinthe

```
type state
val success: state -> bool
type move
val moves: state -> move list
val move: state -> move -> state
```

```
let rec find s =
  success s || trymove s (moves s)
and trymove s = function
  | []     -> false
  | m :: r -> find (move m s) || trymove s r
```

sans persistance

avec un état global modifié en place :

```
let rec find () =
  success () || trymove (moves ())
and trymove = function
  | []     -> false
  | m :: r -> (move d; find ()) || (undomove m; trymove r)
```

i.e. il faut **annuler** l'effet de bord (*undo*)

persistance et backtracking (2)

programmes C/Java très simples, représentés par

```
type stmt =
  | Return of string
  | Var    of string * int
  | If     of string * string * stmt list * stmt list
```

exemple :

```
int x = 1;
int z = 2;
if (x == z) {
  int y = 2;
  if (y == z) return y; else return z;
} else
  return x;
```

persistance et backtracking (2)

on veut vérifier que toute variable utilisée est auparavant déclarée
(dans une liste d'instructions)

```
val check_stmt: string list -> stmt -> bool
val check_prog: string list -> stmt list -> bool
```

persistance et backtracking (2)

```
let rec check_stmt vars = function
  | Return x ->
      List.mem x vars
  | If (x, y, p1, p2) ->
      List.mem x vars && List.mem y vars &&
      check_prog vars p1 && check_prog vars p2
  | Var _ ->
      true
```

```
and check_prog vars = function
  | [] ->
      true
  | Var (x, _) :: p ->
      check_prog (x :: vars) p
  | i :: p ->
      check_stmt vars i && check_prog vars p
```

persistance et backtracking (2)

```
let rec check_stmt vars = function
  | Return x ->
      List.mem x vars
  | If (x, y, p1, p2) ->
      List.mem x vars && List.mem y vars &&
      check_prog vars p1 && check_prog vars p2
  | Var _ ->
      true
```

```
and check_prog vars = function
  | [] ->
      true
  | Var (x, _) :: p ->
      check_prog (x :: vars) p
  | i :: p ->
      check_stmt vars i && check_prog vars p
```

persistance et backtracking (3)

programme manipulant une base de données

mise à jour complexe, nécessitant de nombreuses opérations

avec une structure modifiée en place

```
try
  ... effectuer l'opération de mise à jour ...
with e ->
  ... rétablir la base dans un état cohérent ...
  ... traiter ensuite l'erreur ...
```

programme manipulant une base de données

mise à jour complexe, nécessitant de nombreuses opérations

avec une structure modifiée en place

```
try
  ... effectuer l'opération de mise à jour ...
with e ->
  ... rétablir la base dans un état cohérent ...
  ... traiter ensuite l'erreur ...
```

persistance et backtracking (3)

programme manipulant une base de données

mise à jour complexe, nécessitant de nombreuses opérations

avec une structure modifiée en place

```
try
  ... effectuer l'opération de mise à jour ...
with e ->
  ... rétablir la base dans un état cohérent ...
  ... traiter ensuite l'erreur ...
```

persistance et backtracking (3)

avec une structure persistante

```
let bd = ref (... base initiale ...)
...
try
  bd := (... opération de mise à jour de !bd ...)
with e ->
  ... traiter l'erreur ...
```

interface et persistance

le caractère persistant d'un type abstrait n'est pas évident
la signature fournit l'information implicitement

structure modifée en place

```
type t
val create : unit -> t
val add : int -> t -> unit
val remove : int -> t -> unit
...
```

structure persistante

```
type t
val empty : t
val add : int -> t -> t
val remove : int -> t -> t
...
```

interface et persistance

le caractère persistant d'un type abstrait n'est pas évident
la signature fournit l'information implicitement

structure modifée en place

```
type t
val create : unit -> t
val add : int -> t -> unit
val remove : int -> t -> unit
...
```

structure persistante

```
type t
val empty : t
val add : int -> t -> t
val remove : int -> t -> t
...
```

persistance et effets de bords

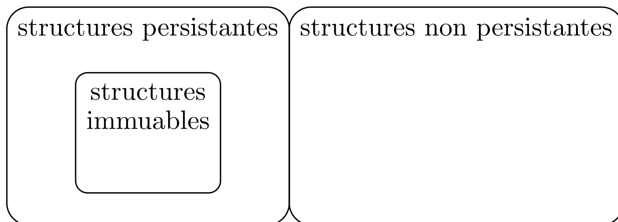
persistant ne signifie pas sans effet de bord

$$\textit{persistant} = \textit{observationnellement immuable}$$

on a seulement l'implication dans un sens :

$$\textit{immuable} \Rightarrow \textit{persistant}$$

la réciproque est fausse

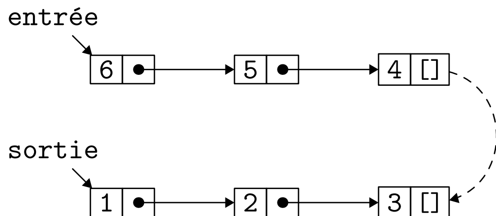


exemple : files persistantes

```
type 'a t
val create : unit -> 'a t
val push : 'a -> 'a t -> 'a t
exception Empty
val pop : 'a t -> 'a * 'a t
```

exemple : files persistantes

idée : représenter la file par une **paire de listes**,
une pour l'entrée de la file, une pour la sortie



représente la file $\rightarrow 6, 5, 4, 3, 2, 1 \rightarrow$

exemple : files persistantes

```
type 'a t = 'a list * 'a list

let create () = [], []

let push x (e,s) = (x :: e, s)

exception Empty

let pop = function
  | e, x :: s -> x, (e,s)
  | e, [] -> match List.rev e with
    | x :: s -> x, ([], s)
    | [] -> raise Empty
```

exemple : files persistantes

si on accède plusieurs fois à une même file dont la seconde liste `e` est vide, on calculera plusieurs fois le même `List.rev e`

ajoutons une référence pour pouvoir enregistrer ce retournement de liste la première fois qu'il est fait

```
type 'a t = ('a list * 'a list) ref
```

l'effet de bord sera fait « sous le capot », à l'insu de l'utilisateur, sans modifier le contenu de la file

exemple : files persistantes

```
let create () = ref ([], [])

let push x q = let e,s = !q in ref (x :: e, s)

exception Empty
```

```
let pop q = match !q with
  | e, x :: s -> x, ref (e,s)
  | e, [] -> match List.rev e with
      | x :: s as r -> q := [], r; x, ref ([], s)
      | [] -> raise Empty
```

exemple : files persistantes

```
let create () = ref ([], [])

let push x q = let e,s = !q in ref (x :: e, s)

exception Empty
```

```
let pop q = match !q with
  | e, x :: s -> x, ref (e,s)
  | e, [] -> match List.rev e with
      | x :: s as r -> q := [], r; x, ref ([], s)
      | [] -> raise Empty
```

récapitulation

- structure persistante = pas de modification observable
  - en OCaml : `List`, `Set`, `Map`
- peut être très efficace (beaucoup de partage, voire des effets cachés, mais pas de copies)
- notion indépendante de la programmation fonctionnelle